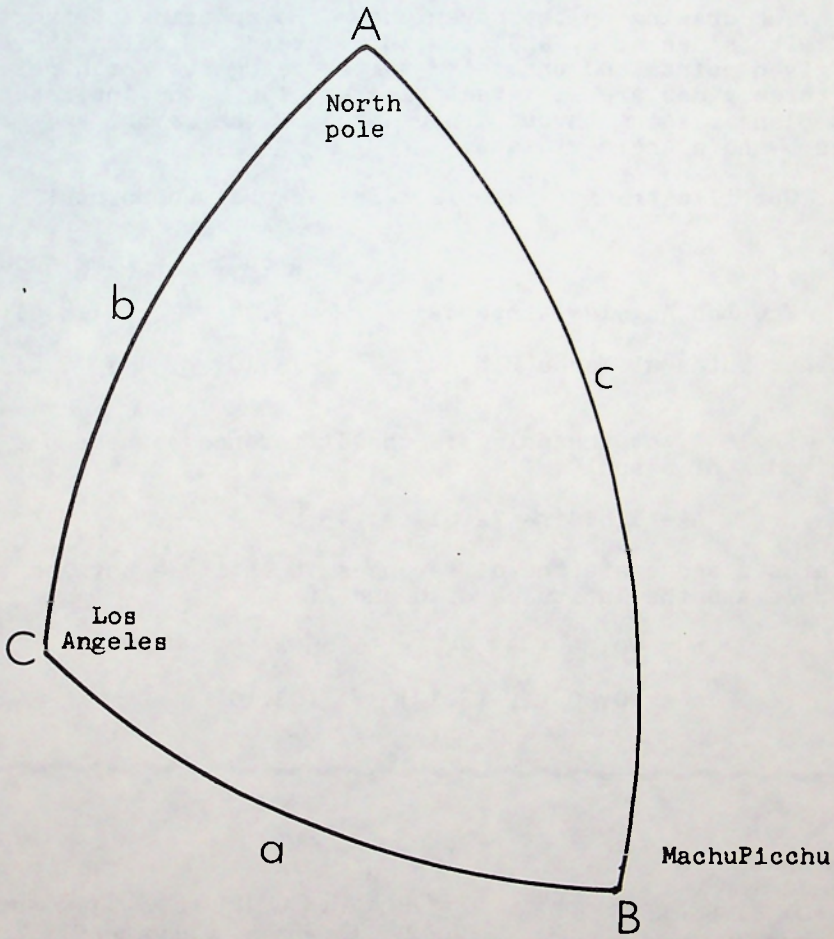
Popular Computing

77

The world's only magazine devoted to the art of computing.

August 1979 Volume 7 Number



How Far To Machu-Picchu?

How Far To Machu-Picchu?

The Problem is an elementary one in navigation; namely, to find the great circle distance between two points on the earth's surface, given the latitude and longitude of the two points.

The following analysis is furnished by Herman P. Robinson.

The drawing on the cover shows the spherical triangle, ABC, with sides a, b, and c, with two vertices being the two given points and the third vertex being the North pole. The three sides are all great circles (i.e., the intersection of a plane passing through the center of the earth) and sides b and c are meridians.

Our illustrative example takes as the two points:

	Latitude	Longitude
The Los Angeles airport :	+33.94	+118.41
The Inca ruins at Machu Picchu:	-13.10	+ 72.61

Angle A, at the pole, is the difference between the longitudes of B and C.

$$A = 118.41 - 72.61 = 45.8$$

The arcs b and c are the differences in latitude between the pole and the latitudes of C and B:

$$b = 90.00 - 33.94 = 56.06$$

$$c = 90.00 - (-13.10) = 103.10$$

Publisher: **Audrey Gruenberger**
Editor: **Fred Gruenberger**
Associate Editors: **David Babcock**
Irwin Greenwald
Patrick Hall
Contributing Editors: **Richard Andree**
William C. McGee
Thomas R. Parkin
Edward Ryan
Art Director: **John G. Scott**
Business Manager: **Ben Moore**

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $20.50 per year, or $17.50 if remittance accompanies the order. For Canada and Mexico, add $1.50 per year. For all other countries, add $3.50 per year. Back issues $2.50 each. Copyright 1979 by POPULAR COMPUTING.

The arc a is then given by:

$$\cos a = \cos b \cdot \cos c + \sin b \cdot \sin c \cdot \cos A$$

and for our example, this is:

$$\cos a = (.55832)(-.22665) + (.82962)(.97398)(.69717)$$

$$\cos a = -.12654 + .56334$$

$$\cos a = .43680$$

and then $a = \arccos(.43680) = 1.118758$ radians

The 1979 American Ephemeris and Nautical Almanac gives the value of the earth's equatorial radius as 6,378,160 meters, so the distance corresponding to a is 7,135,618 meters, or 4434 statute miles. All of these calculations neglect things like:

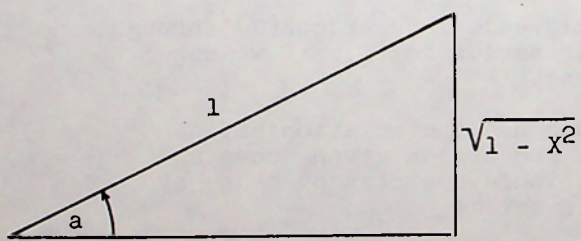
a) The earth isn't a sphere.

b) The Los Angeles airport and the railway station at MachuPicchu differ in elevation by some 8000 feet (2450 meters).

However, the result is probably good to the nearest statute mile.

A program, in BASIC, to perform these calculations is shown. There is also given a table of the coordinates of several other cities, to aid in using the program. Simply insert the latitude and longitude of your own position, and have fun.

The critical part of the code is at line 240, which calculates the arccosine according to the relation:



$$\arccos(X) = \arctan\left(\frac{\sqrt{1 - X^2}}{X}\right)$$

(If BASIC offered the function arccos itself, that would be an even simpler solution.)

```
100 INPUT X1
105 REM X1 LATITUDE ØF C IN DEGREES
110 INPUT Y1
115 REM Y1 LØNGITUDE ØF C
120 INPUT X2
125 REM X2 LATITUDE ØF B
130 INPUT Y2
135 REM Y2 LØNGITUDE ØF B

200 AA = Y1 - Y2
210 B = 90 - X1
220 C = 90 - X2
225 Z = .01745329
226 REM Z CØNVERTS TØ RADIAN MEASURE
227 AA = AA*Z
228  B = B*Z
229  C = C*Z

230 X = CØS(B)*CØS(C) + SIN(B)*SIN(C)*CØS(AA)
240 A = ATN(SQR(1 - X*X)/X)
250 M = A*3963.197
260 PRINT X1, Y1, X2, Y2, M
270 GØTØ 100
300 END
```

Program in BASIC to
calculate great circle distances

Suppose that arccosine is needed (in any program), but the arctan function is not available? What, then, are the possible courses of action?

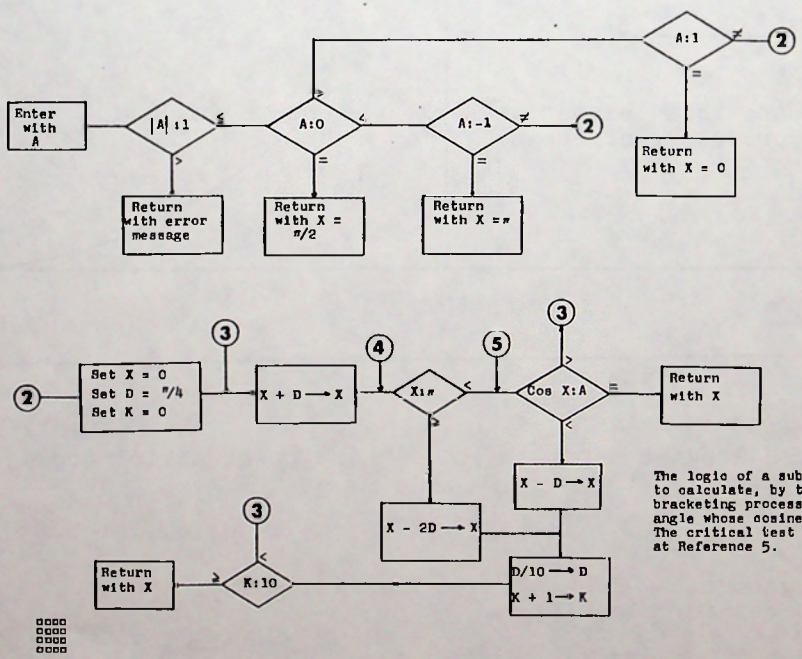
a) We could calculate arccos(X) through a power series in (X), if we can find one.

b) We could use the relation between arccos and arctan given above and then calculate arctan by means of a power series.

c) We could invert the cosine function by bracketing, as was explained back in issue 35 in the essay on bracketing.

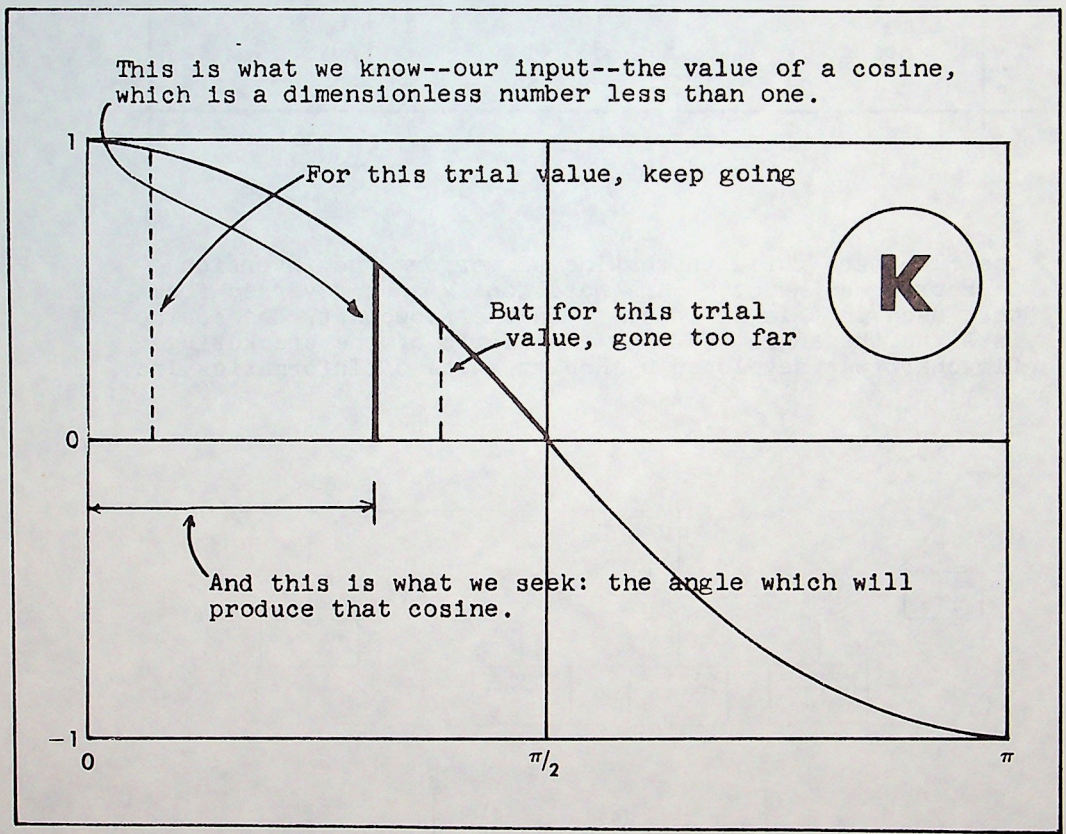
City	Latitude	Longitude
London	+51.5	00.0
New York	+40.75	+74
Los Angeles	+32.94	+118.41
Milwaukee	+43.1	+87.8
Stockholm	+58.6	+18.
Paris	+48.5	- 2.2
Moscow	+55.2	-37.
Hongkong	+22.	-114.
Tokyo	+35.8	-140.
Lima	-12.	+76.9
Santiago	-33.5	+70.5

The flowchart for a subroutine to perform the inversion is reproduced here. (The notation is just reversed from that used in this article. In the flowchart, the cosine is A and the angle is X.) The logic of the bracketing flowchart was developed by Robert White of Informatics Inc.



The logic of a subroutine, to calculate, by the bracketing process, the angle whose cosine is A. The critical test is at Reference 5.

Method (c) can be explained with reference to diagram K. We seek some angle whose cosine is the given value. The function cosine is available to us. If we are using an 8-digit arithmetic system, there are some 15,700,000 possible values of the angle between zero and pi/2, and we could write a loop to try each of them until, for one of them, the cosine agreed with the input value. The bracketing process simply speeds things up enormously, so that only a few dozen (at most) angles need be tried.



Let's look at solution (b). The power series for arctangent is:

$$\text{Arctan}(X) = X - X^3/3 + X^5/5 - X^7/7 + X^9/9 - \cdots$$

which is good for $-1 \le X \le 1$. If X is +1, we have one of the oldest formulas for π:

$$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + \cdots$$

It is also one of the slowest converging series you are likely to meet:

This result first reached	At this term number
.7	4
.78	15
.785	628
.7853	2548

Thus, if the tangent is close to one, the calculation of the angle will tie up the machine for a long time. The arctan series calculation can be speeded up, however, by a simple trick.

$$\tan 2B = \frac{2 \tan B}{1 - \tan^2 B}$$

$$\tan 2B - \tan 2B \tan^2 B - 2 \tan B = 0$$

This is a quadratic equation in the variable (tan B), so we can solve by the quadratic formula to obtain:

$$\tan B = \frac{1 \pm \sqrt{1 + \tan^2 2B}}{-\tan 2B}$$

For example, using the worst case where tan 2B = 1, the transformation above suggests calculating first:

$$\tan B = \frac{1 \pm \sqrt{2}}{-1}$$

$$\tan B = .414213562$$

and this value can be plugged into the same series, with the resulting angle doubled. We now have:

This result first reached	At this term number
.78	1
.785	2
.7853	4
.78539	6
.785398	7
.7853981	8

Thus, for the price of a square root (and our issue number 20 detailed some 13 distinct algorithms for that function), the series calculation for arctan can be greatly speeded up. Whenever arctan is to be calculated by a series, it would be sensible to test X in the program; if X is less than, say, .6, then one could proceed to evaluate arctan(X) directly. If X is greater than or equal to .6, then the half-angle reduction trick will pay off in reduced machine time.

A series for arcsin(X) is:

$$\mathrm{Sin}^{-1}(X) = X + \frac{1\cdot X^3}{2\cdot 3} + \frac{1\cdot 3\cdot X^5}{2\cdot 4\cdot 5} + \cdots + \frac{1\cdot 3\cdot 5\cdots(2N-3)X^{2N-1}}{2\cdot 4\cdot 6\cdots(2N-2)(2N-1)}$$

and, since arcsin(X) + arccos(X) = $\pi/2$, arccos is readily calculated from the above series.

The use of series calculations for function evaluation is fraught with booby traps and dangers, however. One of the many possible troubles is vividly presented in Dorn and McCracken's Numerical Methods, in their Case Study 3. They give a legitimate, well-written, debugged Fortran code to evaluate sin(X) by series, and suggest that the student copy and run it. They even show their results, in both single- and double-precision. They wish to show that the series calculation, applied blindly, leads to ridiculous results, especially for large values of X. When this exercise is assigned to a class, even stranger things happen. The program is to calculate sin(30), sin(390), ..., sin(2190), all of which should be exactly .5. Here are some actual results at 2190° from a class, all of whom supposedly used the code in the textbook on the same machine:

```
   56234.8948955
    7675.65381097
   -1195.776992
  426211.3689352
   57481.9600934
  -28433.21551609
245183776.
         .52630767
       21.56892964
10169188.
-3110400.696926
60114491.
```

Solution Evolution – 3

by Madalyn Babcock

A classic problem given to beginning students in computing is the calculation of some series. The task is relatively simple, well defined, and incorporates all of the basic operations possible in a program; namely, input, calculation, decision-making, looping, and output. An example of such a series is the Taylor series for the calculation of e, the natural logarithm base:

$$e = 2 + 1/2! + 1/3! + 1/4! + \cdots$$

The series is well behaved and converges rapidly (see Table 1).

Table 1

Number of terms versus precision

Number of correct digits in e	Number of terms needed for that precision
10	14
50	43
100	71
250	146
500	255
1000	451
1500	634
2000	809

As a first assignment, most instructors expect programs such as the sample below to be turned in by the members of the class. However, the solution to the problem need not stop there and, in fact, can be expanded into a full semester project. This article traces the steps that a student might take in developing and refining a general program to calculate e to any precision level.

Sample Code

```
100 REM CALCULATIØN ØF E
110 E = 2
120 X = 1
130 FØR N = 2 TØ 16
140 X = X/N
150 E = E + X
160 NEXT N
170 PRINT "E=";E
180 END
```

First, we need a clear statement of the problem to be solved. The problem description should be short and simple, yet touch on each of the major points. The following will do nicely:

> Problem: Design, flowchart, code, and test a program to calculate the value of e to a given number of decimal digits. The number of digits is to be requested by the program, and the range is to be checked (1 to 2000). Use the series
>
> $$e = 2 + 1/2! + 1/3! + 1/4! + \cdots$$
>
> to calculate e. Any appropriate language can be used.

The above problem description is fine for what it is--a brief overview of the problem to be solved. It is not sufficient if taken as a design specification. For that purpose it raises more questions than it answers.

The first task at hand for the student is to flesh out the above description by answering such questions as:

- Which language should be used? If you know only one language, the question is easy to answer. But if you know more than one, how do you go about deciding which is most appropriate? For this article, standard Dartmouth BASIC was chosen because it is simple and can be executed on most systems.

- Is the value of e to be printed, stored, or even be discarded? The problem description doesn't say. We chose to print the final value.

- What should be done if the number of digits requested is out of range? Ignore it, abort the program, print an error message and abort, print an error message and ask again, print an error message and assume some default--or what? Printing an error message and asking again seems the most reasonable approach.

- How do you test such a program? The choice for this problem was to compare the results against published values of e (where do you look for 2000 digits of e?) and also try various precision levels to check the input range testing.

- How detailed should the flowchart be? More on this later.

- How does one "do" high precision arithmetic on the computer? The approach taken for this problem was to represent each number as an array of single digits. The two needed operations of addition and division can then be simulated by loops which process one digit at a time, handling arrays appropriately.

As the student gets deeper into the program, many more questions will arise. It is important to answer these questions as they come up and add the answers to the design specification. These "design notes" are invaluable in testing the program, and they make the final documentation easy to produce since most of it will have already been written. Which brings up another question:

- What documentation is needed? The answer is most often given to us in the form of the minimally acceptable items to be included. But how much should be done? A written problem statement, design specifications, flowchart, program listing, test procedure, and (most importantly) the results are usually sufficient.

Assuming that the basic design work has been done, how does one proceed? For a trivial problem, one can draw the entire flowchart at one time. For non-trivial problems--and problems in the real world are never trivial--the best approach is one of the following:

1) Break the problem into smaller, more manageable problems, and then attack each smaller piece.

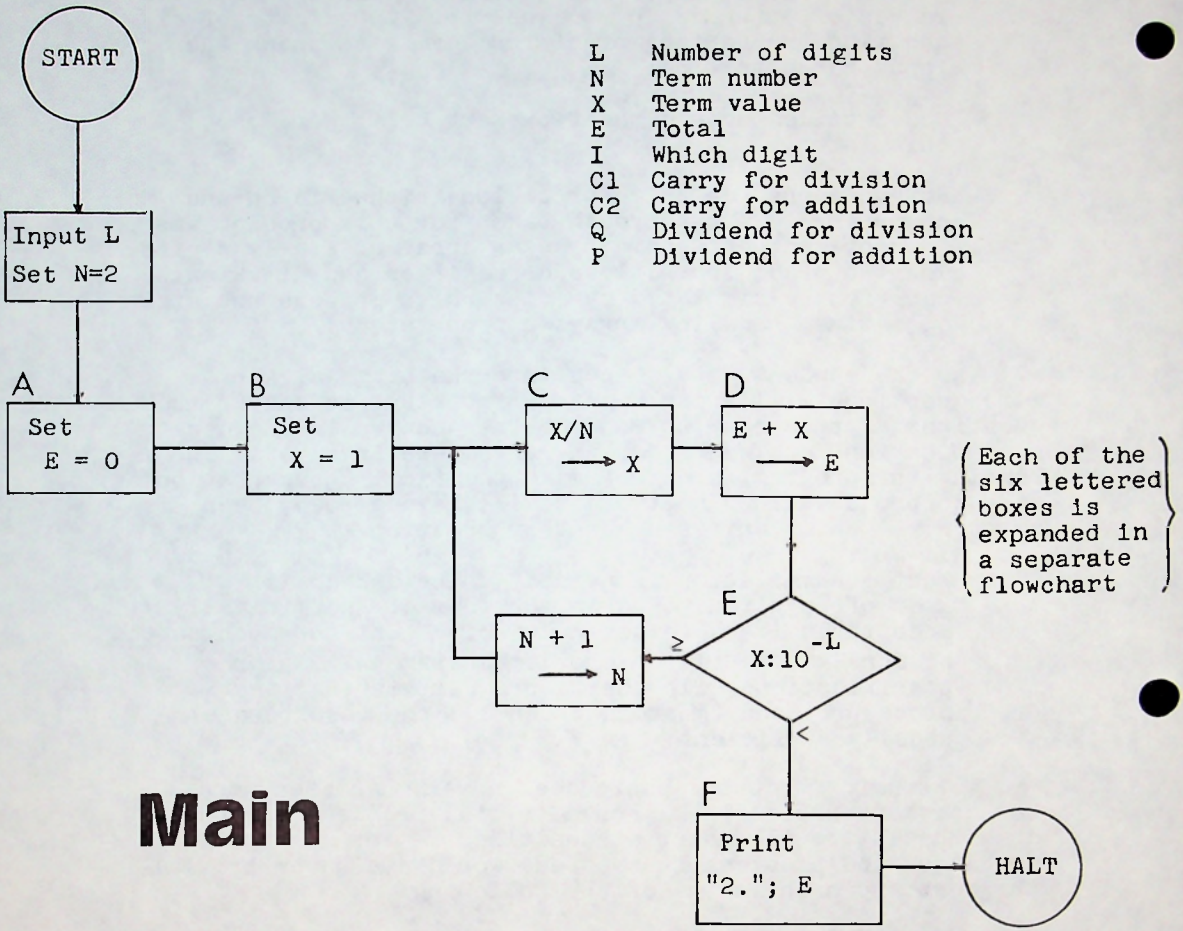
2) Solve a simpler problem and then embellish it until you arrive at the original problem.

For the problem at hand, a combination of both these techniques was used. A simpler problem was flowcharted, then broken into pieces and each of the pieces was then expanded.

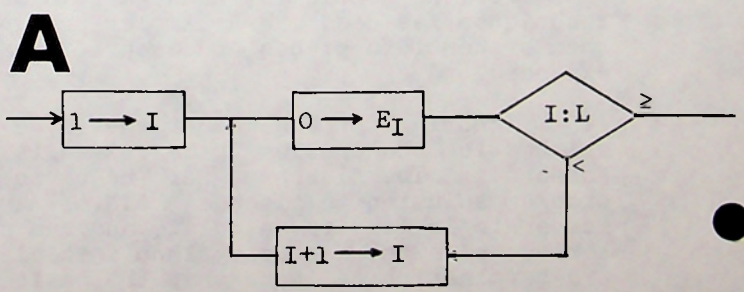
Ignoring for the moment the issue of high precision arithmetic, the problem to be solved is relatively straightforward. The flowchart for the basic logic of this problem solution is given as MAIN. The solution calls for a single loop to calculate and sum each term of the series, plus the necessary loop initialization, testing, and termination. Note that the basic logic of the solution is independent of the precision of the arithmetic operations. The flowchart as given is complete and can be (and, in fact, should be) checked for correctness by the student before going further with the project.

PC77-12

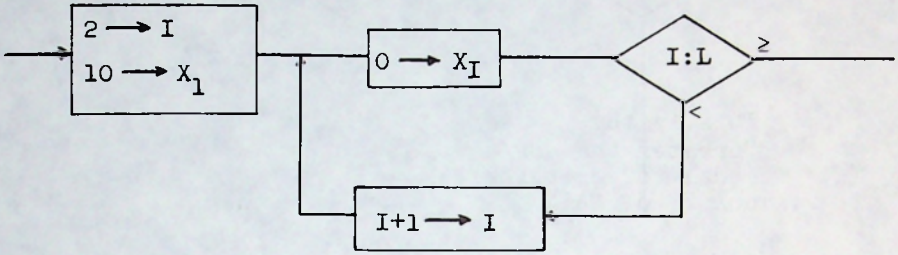
L Number of digits
N Term number
X Term value
E Total
I Which digit
C1 Carry for division
C2 Carry for addition
Q Dividend for division
P Dividend for addition



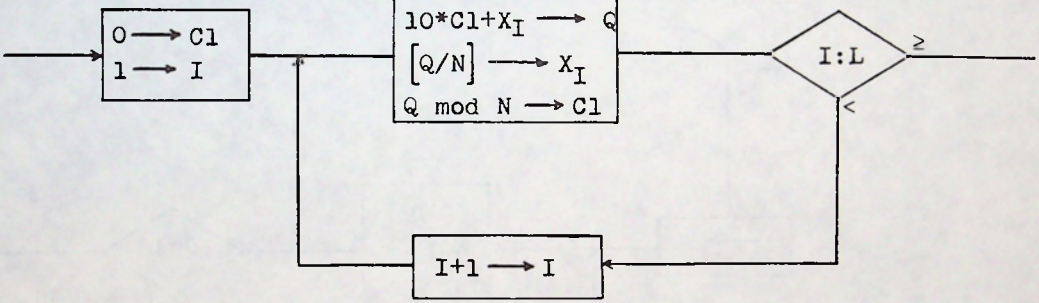
Main



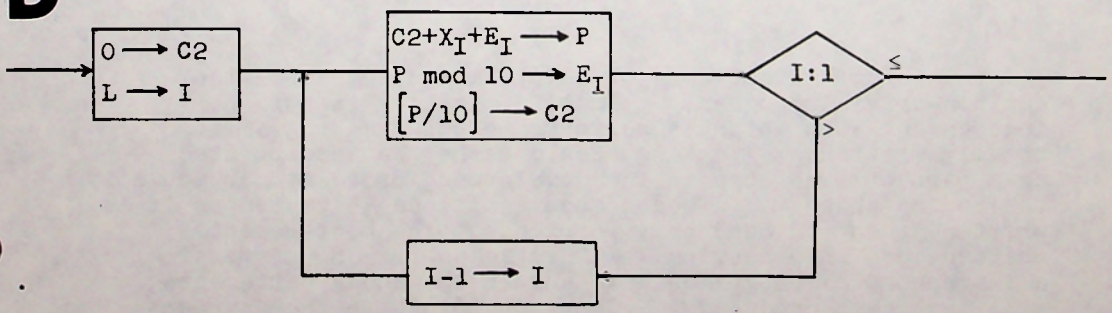
B

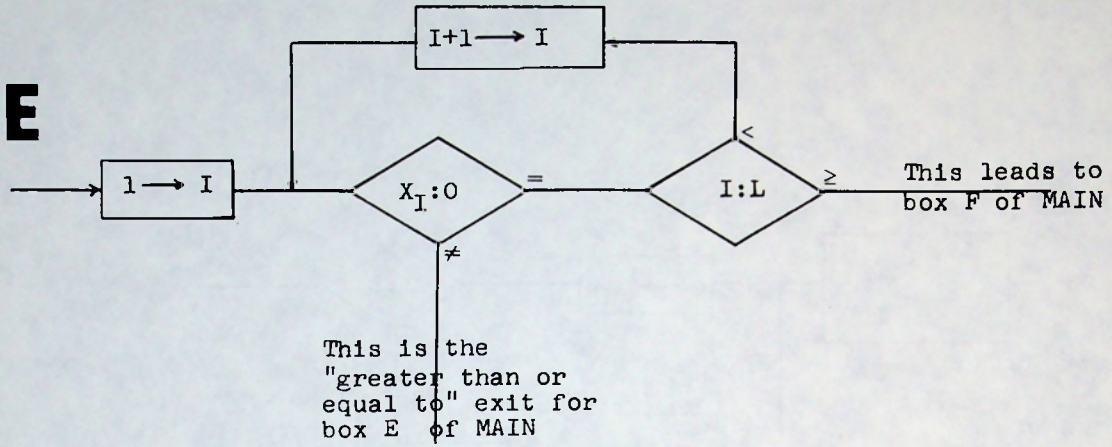

2 → I
10 → X1
0 → XI
I:L
≥
<
I+1 → I

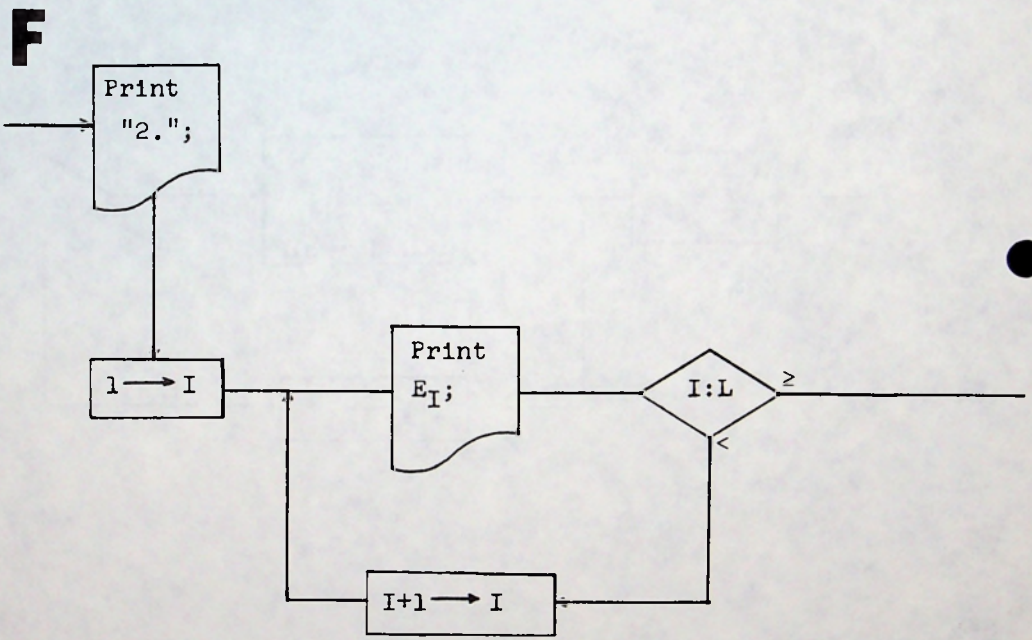
C


0 → C1
1 → I
10*C1+XI → Q
⌊Q/N⌋ → XI
Q mod N → C1
I:L
≥
<
I+1 → I

D


0 → C2
L → I
C2+XI+EI → P
P mod 10 → EI
⌊P/10⌋ → C2
I:1
≤
>
I-1 → I





This brings out a major point in the development of any non-trivial program. That is, testing is not an independent step which is performed after the program is totally written. Testing should rather be incorporated as a part of each step in the development process, particularly the coding phase. If each module of code is tested as it is written, then you need only worry about the most recently written code when testing. This keeps the job of testing a large complex program simple and in the end a better program will have been produced. Contrast this with the approach of waiting until the entire code has been written to begin testing.

Where do you begin to test? How can you be sure that you have tested all possible code paths? (A large program with one decision in each of 20 code modules would result in over a million code paths to test. If each module is tested separately, then only 2*20 tests need to be made.) If the program doesn't work, where do you begin looking for the trouble? In addition to all this, if testing is left until the end to be done, it is usually hurriedly and incompletely done.

...back to the problem at hand. Once the broad brush flowchart has been checked out (either by hand or with a quick 12-line program) we can proceed to add the details of high precision arithmetic to it. The approach taken for this problem was to first identify those boxes on the flowchart which are precision dependent. When they were identified, each of the labelled boxes was in turn treated as a new problem which was then analyzed, flowcharted, coded, and tested. The results of this step are the 6 "sub" flowcharts shown, each of which is designed to replace exactly its corresponding box on the MAIN flowchart.

All the remains to be done in order to produce a correct running code (at least in theory) is to integrate the pieces of code into a single program. In practice it is never that simple. Once consolidated, the program usually fails to run correctly the first time. However, debugging the program is easy if each of the smaller pieces of code were previously checked out. The troubles are then isolated to the "interfaces" between code modules.

That is, a section of code was written assuming that a previous module had done something (like calculate a value) which it didn't do, or at least didn't do it the way in which the following code thought that it should. To find the source of the trouble, analyze each subsection of code as it appears within the whole program. For each module, answer the following question: What assumptions does this piece of code make about what has gone on before it, and did those things indeed happen? The error usually turns out to be a missing line of code or a misspelled variable name in one of the modules. Even those troubles can be reduced if the "inputs" and "outputs" of each module are carefully specified when designing the program.

```
10  REM CALCULATIØN ØF E
20  REM MADALYN BABCØCK
30  REM APRIL 1979
40  REM THIS PRØGRAM CALCULATES E TØ 2000 DIGITS (MAXIMUM)
50  REM IT SIMULATES HIGH PRECISIØN DECIMAL ARITHMETIC
60  REM THE ALGØRITHM USED IS E = 2 + 1/2! + 1/3! +...

100 REM INITIALIZE
105 PRINT "CALCULATIØN ØF E"
110 PRINT
115 PRINT
120 PRINT "HØW MANY DIGITS ØF E TØ CALCULATE";
125 INPUT L
130 IF L<1 THEN 140
135 IF L<=2000 THEN 155
140 PRINT
145 PRINT "THE RANGE ØF DIGITS IS 1 TØ 2000."
150 GØTØ 115
155 L = L + 5
160 N = 2
165 DIM X(L),E(L)
170 DEF FNM(X,Y)=X-INT(X/Y)*Y

200 REM SET THE E ARRAY TØ 0
210 FØR I = 1 TØ L
220 E(I) = 0
230 NEXT I

300 REM SET THE X ARRAY TØ 1
310 X(1) = 10
320 FOR I = 2 TØ L
330 X(I) = 0
340 NEXT I

400 REM CALCULATE X
410 C1 = 0
420 FØR I = 1 TØ L
430 Q = 10*C1+X(I)
440 X(I)=INT(Q/N)
450 C1+FNM(Q,N)
460 NEXT I

500 REM CALCULATE E
510 C2=0
520 FOR I = L TØ 1 STEP-1
530 P=C2+X(I)+E(I)
540 E(I)=FNM(P,10)
550 C2=INT(P/10)
560 NEXT I

600 REM CØMPARE X TØ 10  (-L)
610 FØR I = 1 TØ L
620 IF X(I)<>0 THEN 650
630 NEXT I
640 GØTØ 700
650 N=N+1
660 GØTØ 400

700 REM PRINT E
710 PRINT
720 PRINT
730 PRINT "E=";
740 PRINT "2.";
750 FØR I = 1 TØ L-5
760 PRINT E(I);
770 NEXT I
780 PRINT

800 END
```

Table 2

Execution Times*

Number of digits calculated	Original code	With improvement one	With improvement two
10	7 sec.	6 sec.	3 sec.
50	69 sec.	59 sec.	28 sec.
100	215 sec.	149 sec.	83 sec.
500	$1^h0^m28^s$	40^m17^s	22^m5^s
1000	$4^h33^m25^s$	$3^h20^m42^s$	$1^h16^m34^s$
2000	$12^h40^m52^s$	$9^h16^m41^s$	$4^h30^m32^s$

*Times given are for the Apple II using its integer BASIC interpreter and are approximate.

Improvement One

```
162 Z=1
420 FØR I = Z TØ L
520 I=L
560 I=I-1
570 IF I >=Z THEN 530
580 IF C2<> 0 THEN 530
610 FØR I=Z TØ L
655 Z=I
```

Improvement Two

```
400 REM CALCULATE X AND E
455 E(I)=E(I)+X(I)
500 |
510 |
520 |
530 }          |Delete
540 }          | these
550 |          | lines
560 |
570 |
580 |
640 GØTØ 665
665 REM FIXUP E FØR PRINTING
670 C2=0
675 FØR I=L TØ 1 STEP-1
680 P=E(I)+C2
685 E(I)=FNM(P,10)
690 C2=INT(P/10)
695 NEXT I
```

The project for the most part is now complete, but three remaining steps need to be taken.

1) Tests need to be made on the total program. The program appears to run correctly but it might still have some undiscovered "interface" errors. Exhaustive testing is not required since each module was independently and thoroughly tested. These last tests should include the limit cases, several of the key error conditions, and a couple of the normal cases. (Note: we are assuming that each of the subsections was error free, which may not be the case. If it becomes apparent that there is an error in one of the modules, do NOT continue to test it within the whole program. Instead, test it independently of the other code. This is easier, and also eliminates funny side effects from the other code which tend to mask the true problem.)

2) This step is optional and deals with improving the code. In order to break the original problem into pieces, and then code and test each piece, we were forced to make each module of code as independent from the other modules as possible. In a few cases, this results in needless duplication of code between modules. In other cases, one module could have done something more (which would be easy for it to do and difficult for another module which had to do it) but didn't because more "interface" assumptions would have to be made. In both of these cases, once the whole program is together and tested, modifications and improvements can be made to the code. (Note that this corrupts somewhat the independence of the code modules.)

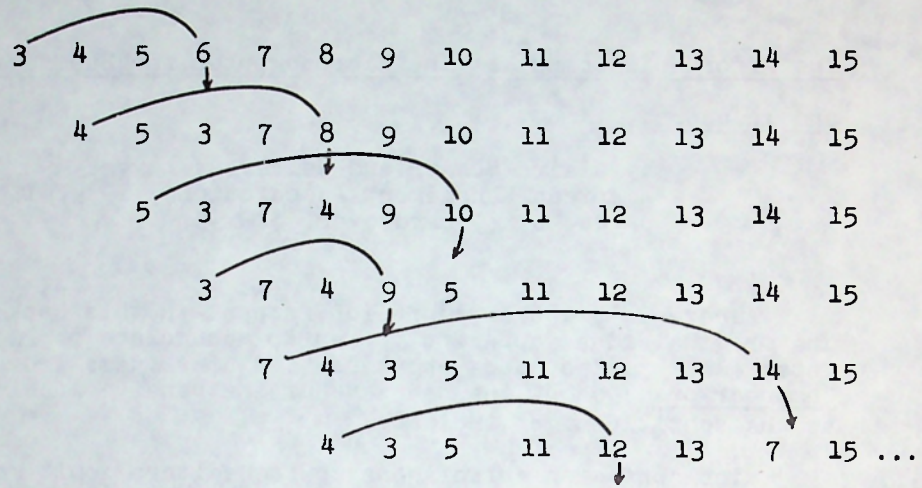
An example of the second case is the program we have developed here. Two code improvements can easily be made which substantially improve the execution time of the program (see Table 2). The first capitalizes on the leading zeros in X. The module labelled E can easily keep track of the leftmost non-zero digit in X. With that information, the C, D, and E modules have to do less work and thus run faster.

The second code improvement is more subtle. The C and D modules could be combined (saving much loop overhead) if it weren't for the fact that carries from the division propagate from left to right while the addition carries go from right to left. Since the "true" value of e is not needed until the final printout stage, we can delay doing the addition carries until near the end. Because of this, we can combine modules C and D by adding another loop to propagate all of the addition carries just before the printout.

3) Documentation. No job is done until the paperwork is finished. However, if the appropriate notes were made along the way, all that needs to be done now is to bring together all the documentation.

There are many ways to approach a project. This article has presented one method which works well by tracing the evolution of a project.

Problem 196 (issue 55) was KNOCKOUT: Take all the positive integers, starting with 3. At each stage, let the leading number be K. Extract the number that is K numbers away from the leader, print it, and move the leader to that position. The first stages are:



And so on. The first 80 terms of the output are shown:

1	6	17	38	33	70	49	54	65	134
2	8	18	40	34	13	50	104	66	136
3	10	19	24	35	74	51	106	67	72
4	9	20	33	36	57	52	19	68	105
5	14	21	46	37	42	53	110	69	142
6	12	22	27	38	60	54	84	70	25
7	4	23	50	39	82	55	20	71	146
8	15	24	39	40	45	56	87	72	111
9	22	25	30	41	86	57	118	73	78
0	5	26	56	42	88	58	63	74	152
1	26	27	58	43	16	59	122	75	154
2	21	28	11	44	69	60	93	76	81
3	18	29	62	45	94	61	66	77	158
4	32	30	48	46	17	62	128	78	160
5	34	31	36	47	98	63	130	79	28
6	7	32	51	48	75	64	23	80	123

```
100 DIM A(600)
110 FØR I = 1 TØ 600
120 A(I) = I + 2
130 NEXT I
140 B = 503
150 N = 1

200 S = A(1)
210 PRINT N, A(S+1)
220 A(S+1) = S
230 FØR I = 1 TØ 599
240 A(I) = A(I+1)
250 NEXT I
260 A(600) = B
270 B = B + 1
280 N = N + 1
290 GØ TØ 200
```

BASIC program to implement KNOCKOUT

S is an intermediate value (i.e., temporary storage)

This scheme is good for at least the first 500 terms of the KNOCKOUT sequence.

BOOK REVIEW

PC77-20

The Affordable Computer: The Microcomputer for Business and Industry

by Claire Summer and Walter A. Levy
American Management Associations, 1979, 179 pages, 6 x 9 hardcover, $12.95

There is a lot of confusion rampant in this book. The original idea, apparently, was to accumulate published material (articles, book chapters, and news items from Computerworld) about the new wave of inexpensive machines, as the subtitle might indicate.

The confusion arises because the editors don't know a mini from a micro, or the price and capability ranges of either. The opening chapter (which has the title "Introduction to Minicomputers") begins:

> Twenty years ago, a high school boy would save up his earnings from delivering newspapers and buy a bicycle. Today, for the same money, he can walk into a local store and buy a computer

which is an incredible statement to make in such a book. Every word of their opening chapter relates to microcomputer (even to the point of mentioning computer stores that will assemble computers for you), but almost all of the rest of the book relates to minicomputers--and the editors were not, apparently, aware of the distinction. The news items (pages 64 through 157) refer to these machines, among others

MITS Altair IBM 32 H-P 21MX Ultimacc
Data General Nova Data General Eclipse
DEC 340 DEC PDP-15

of which just one is usually rated a micro.

Of the four chapters which are not small news items, one ("Basic Principles of Automated Accounting") is a general low-level tutorial not relating directly to any machine. Another ("Claudius: A Microcomputer Order Processing System") relates to a microcomputer marketed only in the United Kingdom. The other two chapters do relate to microprocessors, but to specialized machines which are not generally available. The 14-page glossary by Walter Levy is the only part of the book that is original and well done.

In sum, the American Management Associations have produced a misleading, mislabelled, and generally worthless book. At the price, it constitutes a flagrant consumer rip-off.